TP2 Caml : PRISE EN PIED

Sujet à finir pour le dimanche 28/09/2014.
À rendre par mail à aurelie.lagoutte@ens-lyon.fr

Mode d'emploi Emacs, complément

Il est important de bien indenter son code (c'est-à-dire laisser des espaces en début de ligne de façon à aligner intelligemment certaines lignes de code, par exemple pour le filtrage, pour le début et la fin d'une boucle, etc...). Ceci est vrai dans tous les langages. Pour vous aider à faire cela correctement en Caml, Emacs propose de le faire automatiquement pour la ligne en cours lorsque vous tapez sur TAB. Pensez-y, c'est pratique ! Cela permet parfois de détecter des erreurs avant même la compilation, si l'indentation automatique ne correspond pas à ce que vous espériez (oubli d'un point-virgule, ou d'une parenthèse, etc...).

Environnement, suite

Il est possible d'utiliser `oCaml` en mode compilé. Éditez un fichier `hello.ml`, contenant la commande `print_string "Hello World!"; print_newline();;`. Dans le terminal, compilez puis lancez le programme :

```
ocamlc -o hello hello.ml
./hello
```

Le programme `oCaml` compilé exécute toutes les commandes du code source. Testez le code source suivant :

```
print_string "What is your name? ";;
let s = read_line();;
print_string "Hello world!\n";;
print_string "Welcome to "^ s;;
print_newline();;
5+12;;
print_newline();;
let x = 5+12;;
print_int(x);;
print_newline();;
```

Exercice 1 *Utilisez le code de la suite de Fibonacci du TP1, pour obtenir un fichier exécutable qui lit un entier, et renvoie le n-ième terme de la suite de Fibonacci. On utilisera la fonction* `int_of_string: string -> int` *qui transforme une chaîne de caractère contenant un entier en l'entier lui-même (par exemple,* `int_of_string "34"` *renvoie* `-: int = 34`*). On peut decliner cette formule à volonté :* `float_of_int`, `string_of_int`, *...*

1 Construction de types

Un nouveau type peut être défini à l'aide de la commande suivante :

```
type couleur =
 |Rouge
 |Jaune
 |Vert
 |Bleu ;;

Rouge ;; (* les constructeurs doivent commencer par une majuscule *)

let est_couleur_primaire=function
 |Vert->false
 |_->true;;
```

Les constructeurs peuvent être accompagnés d'arguments ou être récursifs :

```
type nombre =
 |Int of int
 |Real of float ;;

Int(3) ;;
Real(5) ;;

type liste =
 |Liste_vide
 |Cons of (int*liste) ;;
```

Exercice 2 (Pattern matching sur types construits) *En réutilisant les types définis ci-dessus, écrivez une fonction* `somme : nombre -> nombre -> nombre`*. Écrivez également une fonction* `longueur:liste -> int`*.*

2 Retour sur les listes

2.1 Amusettes

Revenons aux listes traditionnelles. Lorsque deux cas du filtrage (aussi appelé *pattern-maching*) doivent renvoyer la même réponse, on peut les accoler sur la même ligne comme en témoigne l'exemple suivant :

```
let au_moins_deux = function
 | [] | [_] -> false
 | t1::t2::q -> true;;
```

Remarquez en revanche que l'exemple suivant renvoie une erreur. Comprenez-vous pourquoi ?

```
let au_moins_deux = function
 | [] | [a] -> false
 | t1::t2::q -> true;;
```

Exercice 3 1. *Écrivez une fonction* `extremum: 'a list -> 'a * 'a` *qui renvoie le couple (min, max) des élements de la liste.*

2. *Écrivez une fonction* `deuxieme: 'a list -> 'a` *qui renvoie le deuxième plus grand élément de la liste. On pourra s'aider d'une fonction auxiliaire qui renvoie une information plus forte.*

Exercice 4 1. *Écrivez une fonction* `map: ('a -> 'b) -> 'a list -> 'b list` *telle que* `map f [a1; ... ; an]` *renvoie* `[f(a1); ... ; f(an)]`.

2. *Texte à trous :*

```
map ........... [4; 7; -5 ; 2; 0];;
- : int list= [6; 9; -3; 4; 2]

map (fun n-> fibo n) ............. ;;
- : int list = [1; 2; 3; 5; 8]
```

2.2 Tris

Exercice 5 (Tri par insertion) *Écrivez une fonction* `insere: 'a -> 'a list -> 'a list` *telle que* `insere a liste_triee` *insère l'élément* `a` *au bon endroit dans la liste* `liste_triee` *que l'on suppose déjà triée. En déduire une fonction récursive* `tri_insertion: 'a list -> 'a list` *qui trie une liste en insérant les éléments les uns après les autres dans la partie déjà triée de la liste (la récursivité nous permettra d'avoir seulement le premier élément à insérer).*

Exercice 6 (Tri par sélection) *Écrivez une fonction* `select_min: 'a list -> 'a * 'a list` *telle que* `select_min l` *renvoie le couple* `(a,reste)` *où* `a` *est l'élément minimum de* `l`*, et* `reste` *est une liste contenant tous les éléments de* `l` *sauf* `a`*. Elle devra fonctionner en un seul parcours de la liste. En déduire une fonction récursive* `tri_selection: 'a list -> 'a list` *qui trie une liste en sélectionnant le minimum de la liste, en le plaçant au début, puis en recommençant avec le reste.*

Exercice 7 (Tri rapide) *Écrivez une fonction* `coupe_pivot: 'a -> 'a list -> 'a list * 'a list` *telle que* `coupe_pivot a l` *renvoie le couple* `(l1,l2)` *où* `l1` *contient tous les éléments de* `l` *qui sont plus petits que* `a`*, et* `l2` *contient tous les éléments de* `l` *qui sont plus grands que* `a`*. Elle devra fonctionner en un seul parcours de la liste. En déduire une fonction récursive* `tri_rapide: 'a list -> 'a list` *qui trie une liste en choisissant le premier élément comme pivot, qui divise la liste en deux morceaux (les plus petits que* `a` *et les plus grands que* `a`*), qui place* `a` *au milieu et qui recommence récursivement.*

Exercice 8 (Tri à bulle) *Écrivez une fonction* `remonte_max: 'a list -> 'a list` *qui fait remonter le maximum de la liste à la fin de celle-ci, en n'échangeant que des éléments successifs. On remarquer que sur une liste triér, cette fonction fait remonter le premier élément de la liste à sa bonne place dans la liste. En déduire une fonction récursive* `tri_bulle: 'a list -> 'a list` *qui trie une liste en faisant remonter chaque élément "dans sa bulle".*

3 Fonctions sur les entiers

Exercice 9 *Écrivez une fonction* `pgcd: int -> int -> int` *qui renvoie le pgcd de deux entiers. Pour réduire le nombre de cas dans le filtrage, on pourra filtrer sur* `(min a b, max a b)`.

Exercice 10 – *Écrivez une fonction* `composition_repetee: ('a -> 'a) -> int -> 'a -> 'a` *telle que* `composition f n a` *renvoie* `f ( f ( ... f(a)))` *où* `f` *est appliquée n fois.*

– *Que renvoie l'instruction* `map (fun n-> composition_repetee (fun x->2*x) n 1) [1;2;3;4;5];;` *et pourquoi ?*